Event Manipulation for Discrete Simulations Requiring Large Numbers of Events

Ernst G. Ulrich
Digital Equipment Corporation

The event-manipulation system presented here consists of two major parts. The first part addresses the familiar problem of event scheduling efficiency when the number of scheduled events grows large. The second part deals with the less apparent problem of providing efficiency and flexibility as scheduled events are accessed to be executed. Additional features and problems dealt with include the proper handling of simultaneous events; that certain events must be created, scheduled, and executed at the same points in simulated time; that infinite loops caused by the concatenation of such "zero-time" events are possible and must be diagnosed; that maintaining various event counts is practical and economical; and that a capability for handling "time-displaceable" events is desirable and possible.

Key Words and Phrases: simulation, discrete systems simulation, discrete event simulation, time-flow mechanisms

CR Categories: 3.65, 3.74, 4.22, 8.1

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Author's address: Digital Equipment Corporation, 146 Main Street, Maynard, MA 01754.

© 1978 ACM 0001-0782/78/0900-0777 \$00.75

1. Introduction

The scheduling algorithm described here is an improvement over a previous version [9] and combines the standard linear list [1, 3, 4] scheduling method with the more recent time-mapping [7, 8, 6] method. The basic strategy used is break up the (potentially) very long standard-event list into many short lists [7, 6, 10, 11, 2] for which event scheduling can be done more efficiently. The scheduling algorithm uses a "converging lists" data structure, with lists originating in a time-mapping table and converging into a single list. This structure facilitates time-mapping scheduling without the necessity to separate "imminent" and "remote" events, and also permits adjustments of the simulation time scale whenever the density of imminent events is consistently high or low.

The scheduling algorithm deals with the problem of simultaneous events [5], and actually gains in overall efficiency as the percentage of simultaneous events increases. Simultaneous events are scheduled into "vertical" sublists suspended from normal "horizontal" event lists.

A crucial and integral part of the event-manipulation system is an "interpretive-branching" mechanism used for event accessing. This mechanism is based on event codes routinely injected into "object" events and on permanent codes located within permanent simulation-control events. This mechanism is flexible and fast because it permits recognition and accessing of control events and object events (distinguished from each other by type and/or attributes) without sequential testing. Adding to this basic flexibility, the event interpretation itself may be modified dynamically by modifying entries within a traverse table used for the interpretation process.

Several related algorithms recently published [10, 11, 2] are characterized by a theoretical preoccupation with the pure problem of event scheduling, and do not consider the problems of event accessing, event counting, simultaneousness, and zero-time event loops.

Furthermore, even the performance results reported for event scheduling (see Figures 4 and 5 in [2]) are not optimal, apparently because the scheduling techniques described are burdened with a high per-event overhead. Specifically, the reported overheads of approximately 100 microseconds (using a Cyber 74!) suggest that at least 100 machine instructions are required to schedule the typical event. With the algorithm described here, fewer than 30 instructions are required. This basic advantage, in conjunction with the features outlined above and to be described in detail, establishes a substantial performance superiority over these algorithms.

2. The "Converging Lists" Data Structure

Figures 1 and 2 illustrate the difference between a conventional linear-list event representation and the converging lists representation. Combining a time-mapping

Fig. 1. Conventional linear-list representation. The current time is $t = 43 \frac{3}{8}$. The minimal time resolution is assumed to be $1/8$ unit.

Fig. 2. Converging lists representation using eight lists and time-mapping table of length eight. Each converging list has 16 time slots; eight for simulation-control events and eight for simulation of object events.

table (time wheel) with multiple linear lists permits the breaking up of a long linear list (Figure 1) into short lists (Figure 2) for which event scheduling can be done quite efficiently. Letting these lists converge at a permanent "threshold" event ($th1$) allows scheduling without separating imminent and remote events. Scheduled events will simply reach their natural destination points, (i.e. imminent time points) before the threshold event or remote time points after the threshold event.

As a simulation progresses and time advances, the time in the threshold event must be updated twice for each scan of the time-mapping table, and is always set to a value that is a full table length beyond the "current" simulation time. This updating process is triggered when the minor threshold events $th2$ or $th3$ are encountered during the event execution, and actually includes updating the threshold time in $th2$ and $th3$ to the same value.

In Figure 2 the time slots available for object events are $0, 1/8, 2/8, \dots$, etc. Different time slots, i.e. $1/16, 3/16, \dots$, etc. are used for simulation-control events. These "privileged" time slots are intended for control events such as print events, halts, status savings, time rollbacks, and the threshold events $th1, th2$, and $th3$.

3. The Event-Scheduling Process

The scheduling process to be used is a normal, forward-directed list-scanning procedure that must be preceded by a time-mapping calculation to establish a time slot for the new event. The time-mapping calculation consists of obtaining a chosen group of embedded digits

Fig. 3. A standard loop and an improved loop for event scanning.

from the calculated event time, and shifting it into a suitable position to serve as an index for the time-mapping table. The actual calculation is based on binary numbers, and simply consists of a shift operation and the removal (masking) of high order bits from the index. For example, a calculated event time of $1655 \frac{3}{8}$ (0011001110111011) is first shifted right to obtain 0000000110011101 and then masked to produce a three bit index 0000000000000101 for a time-mapping table of length eight. Using a binary representation, the length of the time-mapping table must be 2^N "major" time units, and should be large (256, 512, or even 1024 units), so that virtually all scheduled events become imminent-rather than remote events.

A normal list-scanning method to find the time point for event insertion is shown in Figure 3(a). An improvement over this method uses the technique of loop unrolling and is shown in Figure 3(b).

The detailed event-insertion process is straightforward as long as the event time of the new event is not matched by that of another event already contained in the event list. For this simple case the new event is inserted between the two events having a smaller and larger event time than the new one.

Fig. 4. Data structure for horizontal and vertical lists.

However, for lists containing many events, the probability of encountering simultaneous events is large, and a scheduling mechanism taking advantage of this fact may be employed. Figure 4 shows a data structure for the scheduling of simultaneous events which is based on inserting these events into vertical lists attached to the horizontal event lists. This mechanism is operated with the help of event codes which distinguish normal object events from vertical list (vL code) headers. Figure 4 illustrates what happens if a matching event time is found during the scan of the event list. If a matching item is a single event, a vertical sublist must be created with the new event as the second member of this list. If the matching item is a vertical list header, the new event is simply inserted as the last event into this already existing vertical list. The code mechanism described here becomes crucial during the event accessing and execution phase and will be discussed in more detail in the next section.

It should be observed that the initial creation of each vertical list requires overhead. That is, the time needed is approximately twice as long as the conventional substitute of simply advancing along the horizontal list, performing another comparison, and then inserting the matching event. However, this overhead is approximately recovered if two additional events are scheduled at or beyond this point in the horizontal list. After this, all events reaching this point will be scheduled faster than with the conventional method.

The overall efficiency of this scheduling technique compares favorably with other methods. Using the indexed-list method [10] as a point of reference, the following advantages or disadvantages are apparent:

1. Based on a table length of 2^N major time units, the entire time-mapping calculation consists of a simple shift-and-mask sequence. The corresponding calcu-

lation used in reference [10] (page 227, Figure 6, lines 22 through 24) is considerably more complex and much slower.

2. Using the event-scanning method shown in Figure 3(b), scanning is at least 25 percent faster than the standard (Figure 3(a)) method.
3. Regardless of the number of events in a horizontal list, scanning across a "dummy" event [10] is avoided for the scheduling of all imminent events.
4. The periodic scheduling (and removal) of dummy events is avoided here. This is due to the fact that the "dynamic" dummy events used in [10] are replaced by the stationary threshold-event $th1$. This saves overhead, particularly if sparsely populated or empty sublists are unavoidable.
5. A one-directional rather than a two-directional list structure [10] saves additional overhead.
6. Each event insertion requires an extra test to check for the possibility of a matching event time. However, following immediately after a time comparison (Figure 3(b)), this test can normally be done with a single conditional branch.
7. If a match occurs, another test is required to determine if the matched item is a single event or a vertical list (vL code).
8. Using vertical lists means that the number of items appearing in a horizontal list has an upper bound. For example, the arrangement shown in Figure 2 limits the number of object-time slots to eight. This guarantees that the *average* scan crosses four or *fewer* events until the time point for a new (imminent) event is found.
9. Higher overall event-scheduling efficiency is possible by permitting automatic adjustments of the simulation time scale when the event densities of imminent events are persistently high or low (see Sections 7 through 10).

4. Event Accessing and Execution

As distinct from traversing a horizontal linear list, processing becomes more complex when a converging list with attached vertical lists is traversed for the purpose of event "execution." The two basic processing modes which are required are horizontal and vertical event execution.

Horizontal Execution Mode

Each item in a horizontal list must be recognized as one of many possible object-event types, as a vertical-list header, or as one of several possible control events used for simulation control. The recognition process is accomplished by "interpreting" the event code (see Figure 4 and Section 6) in each object event or control event. In other words, the code is used to branch to the program continuation point suitable for the encountered item. The recognition of a normal, "live" event (such as event e_{11} in Figure 4(a)) is typical and leads to the immediate "execution" of this event. Occasionally events are recognized which are either cancelled or blocked. A cancelled event will simply be skipped, and a blocked event will be rescheduled for a later point in time.

Vertical Execution Mode

Recognition of a vertical list (vl code) triggers the temporary termination of the horizontal-execution mode and initiates the vertical execution of simultaneous events contained in this list. Processing in the vertical mode will be mostly identical to horizontal processing. However, events designated by their codes or code attributes as potentially sensitive to other simultaneous events cannot be executed in the same straightforward fashion. The information necessary to process these events will usually be inaccessible to the event-manipulation program, and this is consequently a point where control must be transferred to a user-generated program to perform the necessary analysis. One option possible at this point is to collect all sensitive events and then transfer control to the specified user program.

The processing of a vertical list is terminated when the vertical listheader (vl code) is again encountered. At this point the vertical list has completed its usefulness and could be returned as a whole to a free storage (garbage) list.

From Figure 4 it follows that the vertical-event organization described here is based on a first-in/first-out policy. In other words, simultaneous events are executed in the order in which they were originally inserted into a vertical list. This policy is probably the most practical one for general-purpose simulations, and is compatible with the handling of zero-time events described in the next section. However, flexibility may demand that even this simple policy must be broken. For example, a situation may be envisioned where a control event should be inserted into the top of an existing vertical list in order to trigger a modified traversal of this list.

5. Zero-Time Events

Zero-time events have no duration—i.e. the "residence" time elapsed between event scheduling and event execution is zero. For some simulation problems zero-

time events are unnecessary, but the need for generality makes it desirable to allow such events. It should be noted that zero-time events may create a serious problem, primarily because such events may form infinite zero-time loops which are difficult to pinpoint and wasteful of CPU time. One possible mechanism to detect zero-time infinite loops is based on comparing the values of simulated time periodically, i.e. after executing a fixed number of events. An efficient version of this mechanism is described in Section 8.

6. Control Processing for Event Accessing

Using the time-mapping table, the advance of simulated time consists of scanning this table vertically, traversing each nonempty horizontal list encountered, and traversing all vertical lists suspended from horizontal lists. Specific details are the following:

1. The scanning of the time-mapping table consists of checking each entry against the known address of the major threshold ($th1$) event. Entries pointing to this event signify empty lists to be skipped. Nonempty lists are accessed to perform event execution.
2. After traversing a horizontal list, the major threshold event ($th1$) is normally encountered. The only action to be performed at this point is to return the horizontal list just processed to free storage.
3. Upon reaching one of the minor threshold events, ($th2$ or $th3$) the following actions are required: a) Same as step (2) above. b) The threshold time in all three threshold events is increased to a value which is a full table length beyond the current time. c) The remote list is checked for any events belonging to the just-completed half of the time-mapping table—that is, events having time values less than a full table length ahead of the current time. Usually, there should be none or only a few. If there are any events, they are inserted into this empty half of the time-mapping table. This is done by a special insertion process that moves entire groups of events, including vertical lists, by relinking events and list headers.
4. Upon reaching threshold $th3$, and after performing the items described in (3) above, a new scan of the time-mapping table is started.

As described in Section 4, the execution of events found in horizontal and vertical lists could be identical or slightly different. Distinct executions are realized by using a "traverse table" that consists of two rows, one each for horizontal and vertical traversals. Normally, the event-accessing mechanism is in the horizontal mode, but this mode is switched to vertical when a vl code is encountered. Similarly, when the vl code is encountered during a vertical traversal, the mode is changed back to horizontal.

The correct row and column of the traverse table is accessed by using the current mode (horizontal or vertical) and the current event code as indexes. The infor-

mation obtained for each access is the initial address of program code to be entered. Thus both table rows must contain entries corresponding to all event codes possible for the traversal mode.

The structure of the traverse table permits two useful methods for modifying the interpretation of event codes, thus permitting "run time" modifications of the event execution process:

The first method consists of (usually temporary) modifications of individual table entries. This makes it possible, for example, to count, record, report, screen, reschedule, or inhibit the execution of events of a given type. The modification of table entries will be, of course, triggered by events.

The second method consists of defining and employing traversals (defining entire additional rows in the traverse table) besides normal horizontal and vertical traversals. For example, traversals that count, record, screen, or report every object event being executed are realized with this method.

As a further example, modified traversals that account for the exhaustion of a given "resource" could be set up with this method. All events (recognized by type and/or attributes) attempting to use the exhausted resource, would be interpreted—during the modified traversal—to act appropriately, i.e., by rescheduling or canceling themselves. Also, extra pretraversals or posttraversals which do not execute events are conveniently implemented with this method. For example, if a zero-time loop is highly probable (see Section 8), this may be diagnosed in detail by scanning through already executed object events (a post-traversal) to find common reference points.

Entry into special traversal modes and return to normal modes can be triggered with special object events and normal control events such as vl headers and threshold events.

7. Efficiency of Event Accessing and Control Processing

For event accessing, no direct comparison between this and previous event-handling methods is possible. The reason is that event accessing, obviously less improvable than event scheduling, has been ignored in the literature. This probably means, however, that conventional sequential testing is used to access events. Assuming event organizations of the time-mapping/indexed-list class, some variation of the following method of analyzing the individual event is probably used frequently: Test 1: Test for the termination of an event list by checking a pointer or testing for a termination event. Test 2: Test the accessed event for being a control event or an object event. Test 3: Test the object event for attributes such as live, cancelled, blocked, etc. Test 4: Test for simultaneousness of events. Test 5: Determine the event activity to be performed.

Thus, a total of three to five sequential tests may often be performed before the point of event execution is actually reached. Furthermore, the complexity of tests 4 and 5 may often be considerable.

Clearly, interpreting the current event code by branching via a traverse table represents a faster method of reaching the point of event execution: Typically, four to five instructions will be necessary for the entire interpretation, while sequential testing requires at least two instructions for each simple test, and four or more instructions for each complex test.

The crucial items providing this 2:1 to 3:1 speed advantage are the concentration of information in the event code, and the identical position of this code within object events and simulation-control events. The only significant overhead required to achieve this, the insertion of an event code into each object event, is more than recovered during this process.

Other overhead items, i.e. the updating of threshold events, and the rescheduling and retrieval of remote events, occur infrequently enough to be of little importance for overall efficiency. The overall efficiency of the event-accessing process is also affected by the overhead expended in scanning across periods of time which are absolutely or relatively devoid of activity.

The scanning method used here is faster than the indexed lists method [10] in advancing time when horizontal lists are heavily populated, sparsely populated, or completely empty. The reasons are that the removal and rescheduling of dummy events is unnecessary here, and that empty horizontal lists are scanned across quickly during the vertical scan of the time-mapping table.

Depending upon the characteristic-event distributions of a system being simulated, and assuming that adjustments of the event densities are not possible, the direct overhead savings due to this higher scanning speed may range from negligible to substantial. However, the combined lower overhead due to fast event-accessing and fast scanning often permits a lowering of the event density (Sections 8, 9, 10) and improves event *scheduling* efficiency because horizontal lists are shortened.

8. Measurements and Adjustments to Improve Simulation Efficiency

The techniques described in the preceding sections are of major importance for the efficiency of the event-manipulation system. The techniques to be described in this and subsequent sections are—in terms of efficiency—of little importance for a majority of simulation problems, and become important only if the encountered event distributions are extremely erratic. Although necessary for manipulating remote events, most of the processing discussed after this point will generally require a negligibly small part of the total processing time.

It is assumed here that maintaining two event counts, namely the number of currently scheduled events (e), and the number of already executed events (E), is gen-

Table I

	$E(r)$	$M(r)$	e
Schedule any event			+1
Access single event or first simult. event	+1	+1	-1
Access n th simultaneous event	+1		-1

erally worthwhile. The parameter e is often a vital part of the simulation output because it provides a profile of peaks and valleys in simulated activity. Similarly, the value of E , printed at regular time intervals and at the end of a simulation run is often an invaluable aid toward finding the difference between two otherwise similar simulation runs. Third, the average value of e , labeled $e(av)$ and obtained by random sampling, is a fundamental measure of simulation activity and efficiency.

The event counts e and E are in themselves insufficient to measure the efficiency of the converging-lists method. An additional count needed here is the total number of occupied time points, or major events M , which have occurred during a simulation. These three counts may be updated in parallel—i.e., in three different fields of the same location—by adopting the following method (see Table I): Instead of updating the total values of E and M , two much smaller subtotals of recent values, $E(r)$, and $M(r)$, are used.

Whenever $E(r)$ overflows a relatively small fixed value (such as 255 events)—the totals for E and M —are updated, and the subtotals $E(r)$ and $M(r)$ are set back to zero.

This method, besides assuring small values for $E(r)$ and $M(r)$ and thus permitting parallel updating, also serves a second purpose. It is an aid in the detection of zero-time infinite loops because the current simulation time can be compared for two successive overflows of $E(r)$. If time has *not* advanced between 256 consecutive events, the probability of a zero-time infinite loop is extremely high and control is transferred to a suitable routine verifying this probability (see example in Section 6).

The values of E and M are used to compute two parameters which are necessary to trigger possible adjustments of the simulation time scale. These parameters are the average number of events per horizontal list, and the average number of occupied time points per horizontal list. Both of these are density measurements and their values are $D(e) = E/T$ and $D(m) = M/T$ where T is the elapsed simulation time in terms of major (time-mapping table) increments. Of these, the time-point density, $D(m)$, is the more important parameter for measuring and improving simulation efficiency. In order to achieve optimum efficiency, this value should be adjusted to approximate unity. This value is optimal because event scheduling is most efficient when each horizontal list contains exactly one vertical list, and as many events as possible are scheduled into each vertical list. However, the event density $D(e)$ plays a constructive or destructive role in achieving this optimal situation. If $D(e)$ is sub-

stantially larger than $D(m)$, then each vertical list tends to contain many events and the nearly optimal situation can be approximated. If $D(e)$ is only marginally larger than $D(m)$, then each occupied time point tends to consist of a single event and $D(m)$ should not be adjusted to unity, but a larger value should be used. Longer horizontal lists are desirable in this case because accessing a horizontal list for the purpose of executing a single event would clearly be inefficient due to the overhead involved. Based on the measured and computed parameters discussed above, the following particular adjustments are feasible:

Whenever the average number of currently scheduled events, $e(av)$, remains steady at a low value (between 1 and 10) then the entire converging lists technique becomes less efficient than a straightforward linear-list scheduling/execution technique. This means that scheduling and execution could be done via the remote list, and that the time-mapping computation and the use of the time-mapping table would be unnecessary. However, switching to a linear-list scheduling/execution method is rarely worthwhile because event scheduling and accessing becomes negligible for small numbers of events. Although possible, this option is not further discussed here.

Whenever the time point density $D(m)$ is substantially and steadily below or above one ($D(m) < 9/10$ or $4 < D(m)$), a density adjustment is in order. Depending on the event density $D(e)$, this adjustment must force the density $D(m)$ towards the optimum value of unity or towards a somewhat higher value.

A density adjustment may often improve simulation efficiency, and the adjustment mechanism is described in Section 10. However, before performing an adjustment it will be necessary to consider what happens to remote events before and after a density adjustment. For example, if the number of remote events is statistically negligible, the adjustment should be carried out. If, however, a substantial percentage of scheduled events overflows regularly into the remote region, then any adjustment *decreasing* the event density may be disastrous. The reason is that a minimal adjustment, forcing a lower event density, reduces the time range of the time-mapping table by 50 percent. This means that even more events would overflow after making the adjustment. What is consequently needed before decreasing the event density is a measurement of the average event overflow.

9. Event Trapping and Overflow Measurements

The purpose of the trap (Figure 5) is twofold. First, it is a device to detect and measure the percentage of remote events as these events are scheduled. Second, the presence of the trap will normally speed up the necessary rescheduling of remote events.

Owing to peculiar event distributions, the percentage of remote events may be high and any number of events

Fig. 5. The event-trapping mechanism. The following is done at $t = 43\ 15/16$, i.e. when $th2$ is encountered: 1) The time in $th1$, $th2$, and $th3$ is set to $51\ 15/16$. 2) If the trap is empty, step (3) is omitted. 3) Trapped events are merged into the list of remote events, keeping a count of their number. 4) Remote events with event times less than $51\ 15/16$ are moved into converging lists.

being scheduled could be trapped during each half-scan of the time-mapping table. Therefore, after completing each half-scan and after updating the threshold events, a check of the trap is performed. If there are any trapped events, they are merged into the list of remote events. (See Figure 5 for detailed example.) Depending on the number of trapped events and already-present remote events, total scheduling efficiency is normally superior to scheduling without a trap. The reason is the single merging process which replaces the individual scanning passes necessary for inserting single events. Thus, if event outflow is unavoidable due to peculiar event distributions, the use of the trap will normally lead to better economies than relying on immediate individual-event insertions. A particularly powerful version of the technique would be the combination of the merging process with the insertion of remote events into the empty part of the time-mapping table. (Combination of steps 3 and 4 in Figure 5.) If the outflow of remote events from linear lists is substantial and at the same time rather steady, it should trigger a time-scale adjustment, forcing a higher event density within the time-mapping table.

10. Time-Scale Adjustment

Effecting a time-scale adjustment for the time-mapping mechanism is illustrated in the example shown in Figure 6. For simplicity, the example assumes a doubling of the time scale (desired event density) and also a very short time-mapping table of only eight index positions. It is assumed that the decision to adjust the time scale occurs at $t = 27\ 15/16$, namely upon reaching the threshold event $th2$. This threshold event contains the value $31\ 15/16$ and would normally be updated to $35\ 15/16$ at this time. The following steps will be taken for

Fig. 6. A time-scale adjustment.

the adjustment: 1) The normal updating of the threshold time is suppressed, leaving the value at $31\ 15/16$. 2) The normal loading of any remote events into the empty half of the time-mapping table is suppressed. 3) The threshold code $th3$ is modified to ths (special). After this the simulation will be resumed and will continue until the modified threshold (ths) is encountered. At this point the simulation time is $31\ 15/16$ and the time-mapping table will be completely empty, i.e. any events scheduled but not executed between $t = 27\ 15/16$ and $t = 31\ 15/16$ must be located in the event trap. Now the following additional steps will be taken: 4) The special threshold code is replaced with the normal $th3$. 5) The time in all threshold events is increased by 16 to a new value of $47\ 15/16$. 6) The shift variable used in the shift instruction of the time-mapping index computation is increased by one. (A quadrupling of the time scale would require a value of two.) Now the normal processing will continue. This will first consist of loading the time-mapping table with events from the remote and trap regions. This means all events smaller than the new threshold value ($47\ 15/16$) will be loaded into the table using the modified shift variable and will thus be placed into the correct table location.

11. The Manipulation of Time-Displaceable Events

The daily schedule of a physician provides a good example of a class of problems for which time-displaceable events are desirable. The physician's schedule is created by making patient appointments for fixed points in time, but only the elasticity of this schedule makes it possible to solve problems such as appointment overruns, emergency patients, and no-shows. More formally, the fact that already scheduled events and events-to-be-scheduled can be displaced in time is the reason for the successful functioning of such a system.

Event scheduling and accessing for this class of problems can be done by taking advantage of the time segmentation inherent in the horizontal list organization. A feature essential for realizing this is a bit map representing the physician's time during the time period of a horizontal list. This bit map would be accessible from a "resource" event located at the beginning of a horizontal list, and would be interrogated and updated whenever a new "appointment" event is to be scheduled. The various functions of the bit map would be to indicate available time segments, to prevent overbooking, and to assure a small amount of underbooking to provide execution-time elasticity. During event execution, the bit map would be used for short term adjustments of the schedule when appointment overruns or similar problems occur. Using still-available time segments, these adjustments would consist of advancing, delaying, compressing, or canceling imminent appointments while displacing the fewest number of imminent events. This method, although quite useful for managing a single resource, becomes increasingly powerful if several resources supplying the same service are to be manipulated. For example, the search for an available resource is simplified if a distinction between exhaustively scheduled and still-available resources is made. This can be done, of course, by using event-code attributes to indicate "availability" or "unavailability" of resource events.

12. Conclusions

Of the techniques described here, the control code mechanism is the main vehicle for achieving speed, flexibility, and realization of some of the other techniques. Although requiring the insertion of event codes during event scheduling, this investment is more than recovered during the event execution phase. A good example of the flexibility provided by the code mechanism is the temporary modification of the threshold th_3 to facilitate the time-scale adjustment.

The low overhead provided by permanent threshold events permits the use of relatively short horizontal lists, and thus provides fast event scheduling. For example, the maximum length of the horizontal list can be forced to time units of 8, 4, 2, or 1. This means that the average scan for scheduling a new imminent event cannot encounter more than 4, 2, 1, or 1/2 of the already scheduled items.

It should also be noted that the time range of the time-mapping table can be substantial even if relatively short horizontal lists are used. Recommended values are 16 possible time steps per horizontal list, and a table length of 512 major time units. This amounts to a table range of at least 4096 time units, and means that all events having a residence time shorter than 4096 time units are guaranteed to be scheduled as imminent events.

The vertical list technique used here is based on the viewpoint that the representation of simultaneousness is essential for many simulation problems. This viewpoint

is not shared by all practitioners involved in discrete event simulation, and the alternative usually offered is to declare an extremely fine time resolution that statistically prevents simultaneous events. The following example demonstrates that this alternative is not always applicable: Consider the problem of simulating the flow of automobile traffic at an intersection with stop signs. Assume that the typical driver can only perceive the arrival of his own and other cars at the intersection within an accuracy of 1/5 second.

A natural and sufficiently precise time resolution for this simulation problem is 1/5 of a second, with cars assumed to arrive at the intersection "at" (i.e. approximately) times 0, 1/5, 2/5, 3/5, ..., etc. If two drivers arrive at different times, the one arriving first simply enters the intersection. If two drivers arrive at the same time, they will "see" each other (via the mechanism of simultaneous events) and the one immediately to the right of another one will gain the right-of-way.

This approach will fail when a finer time resolution (such as milliseconds) is used, and when artificially precise arrival times are created at random. For example, a driver arriving at the intersection would not "see" another one arriving only one, two, or three milliseconds later. Therefore, he will always enter the intersection first, even if he comes from the left and should yield the right-of-way.

The vertical list organization provides substantial performance advantages for simulations creating a large percentage of simultaneous events. For example, the logic simulation of digital circuits is characterized by a need for large numbers of simultaneous events, and is usually done by using a time-mapping table with a single vertical list per time slot [8, 6]. Although not quite matching the speed of this specialized technique, the more general technique described here will permit a reasonably fast simulation of digital circuits.

Besides efficiency, the vertical list organization provides options and features which are hard to achieve otherwise. Some of these are the identification of sensitive simultaneous events, the possibility of counting these events, the ordering or counting of events by type or attributes, the possibility of "forcing" nonsimultaneousness of time-displaceable events during event execution by predating or postdating the event time, and fast access to time points immediately beyond a group of simultaneous events.

This last feature is of interest because it helps to solve "last event" and "redundant event" problems often encountered with simultaneous events. Simultaneous events contained in long vertical list have a high probability of generating control events (such as status saving, status printing, time rollback, or halts) which must be executed just after the last simultaneous event is executed. Also, the very length of these vertical lists increases the probability that the same control event (such as "status printing") is generated again and again. The natural place for such control events is the privileged

time slot immediately beyond the current time slot. However, direct scheduling into this slot is inadvisable because the slot may already contain events which should affect or inhibit the scheduling of the new event. Consequently, the scheduling of control events must be done via a conditional scheduling process that analyzes the privileged slot for threshold events, prevents the redundant scheduling of identical events, and performs a priority ordering of event types.

The feature just described would be less convenient and quite costly if simultaneous events were stored in a conventional horizontal fashion and if no privileged time slots exist. First, the overhead wasted for scanning across many simultaneous events to insert the same control event again and again may be considerable. Second, the absence of privileged time slots would complicate the conditional scheduling process because simultaneous object events and control events would be scheduled into the same time slot and would have to be separated.

Although the technique of vertical lists is described in great detail and is the recommended option, it is not an integral part of the event manipulation system. It may be replaced with the conventional technique of horizontal scheduling by replacing four instructions in the scanning loop shown in Figure 3(b). The replacement consists of reducing the "less-or-equal" tests to "less than" tests, and changing the loop exits to branch directly to event scheduling. This option is recommended for a fast implementation of an initial version.

The emphasis placed here on "zero-time" looping is an outgrowth of experience gained in digital logic simulation. For combinational logic it is often quick and expedient to assume zero-time switching delay. If this logic is then expanded into sequential logic, the necessary addition of delay points is sometimes "forgotten", and large amounts of CPU time are often spent to find and eliminate zero-delay loops. It is assumed here that something similar may happen during the development of other systems, particularly during the debugging phase of a simulation problem.

The event-trapping mechanism, in conjunction with the merging of trapped events into the list of remote events, improves scheduling efficiency somewhat when remote events are unavoidable. Also, the trapping mechanism detects the occurrence of remote events and triggers, if appropriate, a time-scale adjustment.

The time-scale adjustment is simple and requires a minimal amount of programming effort. However, for a vast majority of simulation problems, an initial fixed time-scale setting will be sufficient, and no adjustment should be necessary.

A specific version of the described technique is currently being implemented at Digital Equipment Corporation. The purpose of this implementation is the efficient handling of the large number and types of object events arising during the concurrent logic simulation of many single-faulted logic circuits. Some quantitative results achieved with this implementation are reported in [12].

Received August 1975; revised December 1977

References

1. Buxton, J.N., Ed. *Simulation Programming Languages*. North-Holland, Amsterdam, 1968.
2. Franta, W.R., and Maly, K. An efficient data structure for the simulation event set. *Comm. ACM* 20, 8 (Aug. 1977), 596-602.
3. Genuys, F., Ed. *Programming Languages*. Academic Press, New York, 1968, pp. 349-395.
4. Gordon, G. *System Simulation*. Prentice-Hall, Englewood Cliffs, N.J., 1969.
5. Parnas, D.L. On simulating networks of parallel processes in which simultaneous events may occur. *Comm. ACM* 12, 9 (Sept. 1969), 519-531.
6. Szygenda, S., Hemming, C.W., and Hemphill, J.M. Time flow mechanisms for use in digital logic simulation. Proc. Winter Simulation Conf., 1971, pp. 448-495.
7. Ulrich, E.G. Serial/parallel event scheduling for the simulation of large systems. Proc. 1968 ACM Nat. Conf., pp. 279-287.
8. Ulrich, E.G. Exclusive simulation of activity in digital networks. *Comm. ACM* 12, 2 (Feb. 1969), 102-110.
9. Ulrich, E.G. Non-integral event timing for digital logic simulation. Proc. 13th Design Automation Conf., 1976, pp. 61-67. (available from ACM, New York).
10. Vaucher, J.G., and Duval, P. A comparison of simulation event list algorithms. *Comm. ACM* 18, 4 (April 1975), 223-230.
11. Wyman, F. P. Improved event-scanning mechanisms for discrete event simulation. *Comm. ACM* 18, 6 (June 1975), 350-353.
12. Philips, N., and Tellier, J. Efficient event manipulation—the key to large scale simulation. Proc. Cherry Hill Test Conf. (Oct. 1978).